Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems

Angeliki Kritikakou
IRISA-Univ. Rennes 1-ONERA, FR
angeliki.kritikakou@onera.fr

Claire Pagetti
ONERA, Toulouse, FR
claire.pagetti@onera.fr

Christine Rochange
IRIT-Univ. Toulouse, FR
christine.rochange@irit.fr

Matthieu Roy
LAAS-CNRS, Toulouse, FR

Madeleine Faugère
Thales, Palaiseau, FR

Daniel Sylvain Girbal
Thales, Palaiseau, FR

Gracia Pérez
Thales, Palaiseau, FR

ABSTRACT

When integrating mixed critical systems on a multi/many-core, one challenge is to ensure predictability for high criticality tasks and an increased utilization for low criticality tasks. In this paper, we address this problem when several high criticality tasks with different deadlines, periods and offsets are concurrently executed on the system. We propose a distributed run-time WCET controller that works as follows: (1) locally, each critical task regularly checks if the interferences due to the low criticality tasks can be tolerated, otherwise it decides their suspension; (2) globally, a master suspends and restarts the low criticality tasks based on the received requests from the critical tasks. Our approach has been implemented as a software controller on a real multi-core COTS system with significant gains [1].

1. INTRODUCTION

Mixed-critical systems [27] consist in integrating applications with different properties and requirements into a common platform. The platform should provide the required level of dependability, in particular of safety, for each application. The safety level is given by the criticality level of an application, which depends on the consequences on the system if the application doesn't meet its timing constraints. For instance, the criticality level of an avionic application is given by the Design Assurance Level (DAL) model [26].

A high criticality application, e.g. an application of A, B or C level of DAL, requires *predictability* to be provided by the platform, i.e. the ability to compute a safe estimation of the Worst-Case Execution Time (WCET) [28]. The current platforms consist of multi/many-core COTS systems, which are hard, if not impossible, to be predictable [29]. The reason is the complexity of the system architecture, the enhanced mechanisms used to improve the system performance and the lack of the systems' documentation. WCET estimations exist which upper bound the effects of the possible interferences on the system shared resources by assuming full congestion. In this way they are able to provide safe estimations which are required to guarantee real-time response. However, with these techniques a dramatic difference occurs in WCETs of an application estimated 1) when it is executed alone on one core (isolated execution, no interferences) and 2) when other applications are concurrently executed on the remaining cores (maximum load, interferences). This difference may lead to the system unschedulability in case:

$$\mathsf{WCET}_{\mathsf{iso}} < D_C < \mathsf{WCET}_{\mathsf{max}}$$

where $\mathsf{WCET}_{\mathsf{iso}}$ is the WCET of a high criticality application τ_C in isolated execution, D_C is its deadline and $\mathsf{WCET}_{\mathsf{max}}$ is its WCET in maximum load.

1.1 Run-time control for a unique critical task

We have addressed this problem by proposing a safe approach which increases the utilization of the system resources in [17]. The model of our system consisted of $n+1$ independent synchronous tasks $\mathcal{T} = \{\tau_{\mathrm{C}}, \tau_1, \ldots, \tau_n\}$ where τ_{C} is a periodic task of high criticality level (DAL A, B or C) with period T_C and deadline D_C; τ_i are tasks of low criticality level (DAL D or E); one task corresponds to one application.

Our system initially executes all tasks regardless their criticality level. The critical task is modeled by a set of Extended Control Flow Graphs (ECFGs), i.e. control flow graphs with *observation points* where the run-time control is executed. The run-time control regularly checks if the interferences of the low criticality tasks can be tolerated by verifying our safety condition (Eq. 1). When the condition does not hold, the low criticality tasks are immediately suspended to eliminate congestion over the shared resources and to guarantee the critical task's timing constraints. When the task terminates, the low criticality tasks are resumed.

$$\mathsf{RWCET}_{\mathsf{iso}}(x) + \mathsf{W}_{\mathsf{max}} + t_{\mathrm{SW}} \leq D_{\mathrm{C}} - \mathsf{ET}(x) \qquad (1)$$

where $\mathsf{RWCET}_{\mathsf{iso}}(x)$ is the remaining WCET of τ_C in isolated execution from the observation point x until the end, $\mathsf{W}_{\mathsf{max}}$ is the maximum WCET until the next point, t_{SW} is the overhead of suspending the low criticality tasks and $\mathsf{ET}(x)$ is the monitored execution time of τ_C until point x.

[1]The research leading to these results has received funding from the TORRENTS cluster supported by the RTRA STAE and the European FP7-ICT project DREAMS under reference n° 610640.

RTNS 2014 , October 8 - 10 2014, Versailles, France

http://dx.doi.org/10.1145/2659787.2659799.

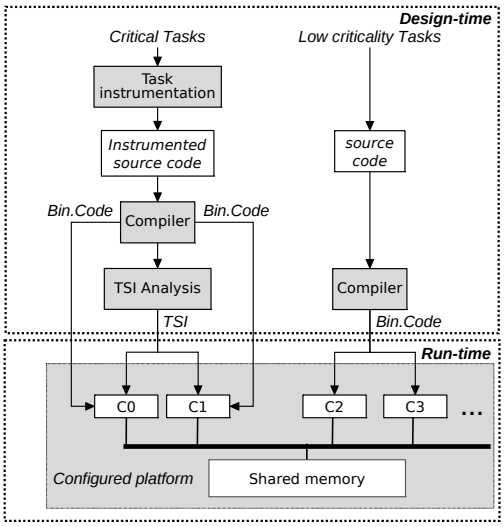


Figure 1: Overview of the proposed approach

1.2 Run-time control for several critical tasks

The contribution of this work is twofold. Firstly, we have extended our approach to several high criticality tasks concurrently executed on different cores with different deadlines, periods and offsets. Secondly, we have implemented our approach as a software controller in a real multi-core COTS system, i.e. the Texas Instrument TMS320C6678 (or TMS in short) [16]. Our objective is the improvement of the system resource utilization by increasing the concurrent execution of the low criticality tasks.

The system model now consists of $p+n$ independent tasks $\mathcal{T} = \{\tau_{C_1}, \ldots, \tau_{C_p}, \tau_1, \ldots, \tau_n\}$ where τ_{C_j} are periodic tasks of high criticality level with period T_{C_j}, deadline D_{C_j}, offset O_{C_j}; τ_i are tasks of low criticality level. A static partitioned scheduling has been applied in which each critical task (which can consists of smaller critical tasks with precedence constraints) is executed on one core and the remaining cores execute the low criticality tasks. In this extension, $\mathsf{WCET}_{\mathsf{iso}}$ is the WCET of τ_C where all the high criticality applications may execute in parallel.

Our methodology applies a design-time part and a run-time part, as shown in Fig. 1. The design-time analysis inserts the proposed run-time controller to the critical tasks. As we focus on implementing our approach on TMS as a software controller, our methodology instruments the critical tasks by refining the *observation points* of the initial ECFGs in [17] with the proposed software control mechanism. The set of critical tasks is now modeled by a set of Instrumented Control Flow Graphs (ICFGs), i.e. control flow graphs enhanced with the software control mechanism. The result is the instrumented source codes of the critical tasks. For the low criticality tasks, no instrumentation is required. Then, at design-time a Timing and Structure Information (TSI) analysis takes place which pre-computes the structure and the timing information required by the run-time control mechanism using the ICFGs per critical task.

At run-time, each critical task executes its own run-time control mechanism, which monitors the ongoing execution time, dynamically computes the remaining WCET of the task in isolated execution and checks its *safety condition*

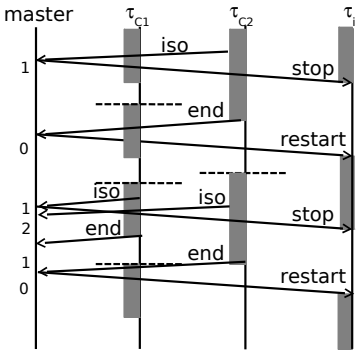


Figure 2: Run-time behavior

to locally decide if the low criticality tasks should be suspended to guarantee that its deadline is met. However, the critical tasks are not responsible for the execution of the low criticality tasks suspension. They send a request to a new entity, the *master*, which has a global view. The master is in charge of collecting the requests of the critical tasks, suspending and restarting the low criticality tasks. The master suspends the low criticality tasks when at least one critical task sends the request for isolated execution, when the safety condition is not satisfied. During execution, the master updates the number of active requests and it restarts the low criticality tasks when all requesters have been executed.

Fig. 2 describes the run-time behavior of our control mechanism through an example with two critical tasks running in parallel. In the first scenario, the safety condition of the critical task τ_{C_2} is violated and thus it sends a request for isolated execution to the master. The master upon receiving this request sets the number of active requests to 1 and suspends the low criticality tasks. Then, the requester (task τ_{C_2}) informs the master that its execution is finished. As the critical task τ_{C_1} has not yet requested isolated execution, no risk exists for the deadline of τ_{C_1}. The master resumes the low criticality tasks (active requests=0). In a later scenario, the critical task τ_{C_1} has also requested isolated execution after the request of τ_{C_2}. The master restarts the low criticality tasks when both tasks have finished. The master is not assigned on the same core with a critical task, as this option will increase the WCET of the critical task due to the received requests. Hence, the master is assigned at a core that executes low criticality tasks.

The remaining of the paper is organized as follows: Section 2 presents the design-time analysis and Section 3 describes the software run-time control mechanism and the master entity. Section 4 presents the software implementation on TMS and several experimental results to evaluate our approach. Section 5 presents the related work on mixed-critical systems. Section 6 concludes this study.

2. DESIGN-TIME ANALYSIS

This section describes the design-time analysis which consists of the instrumentation of the critical tasks described at Section 2.1 and the timing and structure analysis to extract the pre-computed information described at Section 2.2.1. The proposed methodology considers two scenarios for the tasks that are executed on the platform.

Definition 1 (Execution scenarios). *The execution*

scenarios are

1. *Isolated execution (*iso*), where p critical tasks are executed on the platform,*
2. *Maximum load (*max*), where p critical and n low criticality tasks are concurrently executed on the platform.*

2.1 Critical tasks instrumentation

For the instrumentation of the critical tasks, we use as basis the graph grammar presented in [17] and we extended it to describe the proposed software instrumentation of the critical tasks.

DEFINITION 2 (CRITICAL TASK τ_C). *A* critical task τ_C *is a set of functions* $\mathcal{S} = \{F_0, F_1, ..., F_n\}$*, with* F_0 *the main function. Each* function *is represented by an Extended CFG (ECFG).*

DEFINITION 3 (ECFG). *An* extended control flow graph *(ECFG) is a control flow graph extended by adding* observation points*. An observation point is a position where the run-time control is executed. The ECFG of function F is a directed graph* $G = (V, E)$*, consisting of:*

1. *A finite set of nodes V composed of 5 disjoint sub-sets* $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$ *where,*
   - $N \in \mathcal{N}$ *represents a binary instruction or a block of binary instructions,*
   - $C \in \mathcal{C}$ *represents the block of binary instructions of a condition statement,*
   - $F_i \in \mathcal{F}$ *represents the binary instructions of calling a function* F_i *and links the node with the ECFG of the function* F_i*,*
   - IN *is the input node,*
   - OUT *is the output node.*
   - *every node* $v \in V \setminus \{OUT, IN\}$ *has one unique input observation point before the execution of the first binary instruction (the observation point is represented by a lowercase symbol);*
   - start *is the observation point before starting the execution.*
2. *a finite set of edges* $E \subseteq V \times V$ *representing the control flow between nodes.*

The Instrumented Control Flow Graph (ICFG) is derived by replacing the theoretical observation points in the Extended control Flow Graph (ECFGs) [17] by the software run-time control mechanism.

DEFINITION 4 (SOFTWARE RUN-TIME CONTROL). *The* software run-time control mechanism *(Fig. 3) for the critical tasks consists of:*

1. *a condition* C_{RT} *which semantics is:*
   - C_{RT} *= true* $\Longleftrightarrow$ *maximum load scenario,*
   - $\overline{C_{RT}}$ *= true* $\Longleftrightarrow$ *isolated execution,*
   - C_{RT} *= true at observation point* start*.*
2. *if* C_{RT} *is true at* $v_i \in V \setminus \{OUT, IN\}$*, a function call occurs to the decision,* F_D*, which consists of the:*
   - *node* N_C *to monitor the current execution time of the critical task and compute the remaining WCET in isolated execution,* $RWCET_{iso}(x)$*,*
   - *safety condition* C_D *to decide if the low criticality tasks should be suspended. If no suspension occurs, the run-time control returns to the critical task execution,*
   - *node* N_S *to send a request for isolated execution and to turn off the run-time control mechanism by setting* C_{RT} *= false, if decided.*

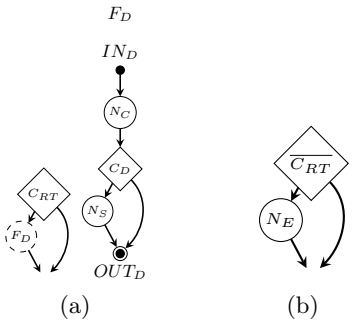


Figure 3: Software control mechanism a) decision and b) end detection

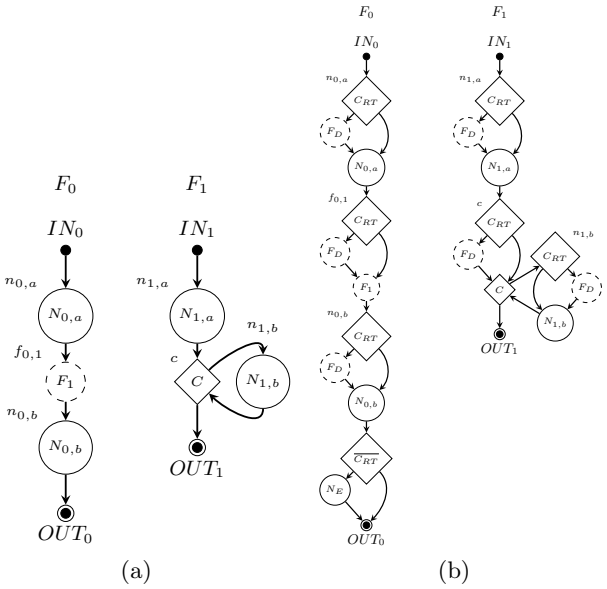


Figure 4: a) ECFGs and b) ICFG for $\mathcal{S} = \{F_0, F_1\}$

3. *if* $\overline{C_{RT}}$ *is true at* $v_i = OUT_0$ *(end of each* τ_C*), the end detection* N_E *is executed to notify the end of the task.*

The Fig. 4(a) presents the ECFGs of a critical task composed of two functions $\mathcal{S} = \{F_0, F_1\}$ and Fig. 4(b) illustrates the ICFGs of the theoretical ECFGs of Fig. 4(a) instrumented by our software run-time control mechanism.

2.2 Timing and Structure Analysis

2.2.1 Structure information

During the execution of any critical task, an observation point may be visited several times due to the loops and the function calls. Hence, we need the structure information of ICFG in order to distinguish between different visits of the same point during run-time execution, i.e. the *nested level*, the *head point* and the *type* of an observation point as defined in [17]. In summary:

- The nested level is: a) 0 for the *start* point, b) 1 for the sequential points between the IN and the OUT of an ICFG and c) increased by 1 when we are entering a loop.
- The head points show when a function has been called and where a loop exists in each ICFG. The head points

Table 1: Structure & timing information of Fig. 4(b)

Observation point x	*level* (x)	*type* (x)	*head* (x)	d(x)	w(x)
Initialization					
start	0	-	-	-	-
F_0					
$n_{0,a}$	1	-	*start*	$d_{start-n_{0,a}}$	-
$f_{0,1}$	1	F_ENTRY	*start*	$d_{start-f_{0,1}}$	-
$n_{0,b}$	1	F_EXIT	*start*	$d_{start-n_{0,b}}$	-
F_1 Fig. 4(a)					
$n_{1,a}$	1	-	$f_{0,1}$	$d_{f_{0,1}-n_{1,a}}$	-
c	1	-	$f_{0,1}$	$d_{f_{0,1}-c}$	w_c
$n_{1,b}$	2	-	c	$d_{c-n_{1,b}}$	-

are: a) the *start* point, for the points of level 1 of the main function F_0, b) the function caller, for the points of level 1 of the remaining functions and c) the condition of the loop, for the points that are inside a loop.

- The type determines: a) the function entry, i.e. the function caller, and b) the function exit, i.e. the observation point just after the return from a function.

To support the run-time control mechanism, we store the structure information in memory. Table 1 provides the structure information for the observation points of Fig. 4(b). The information about the head points is propagated to the timing analysis to compute the partial remaining WCETs.

2.2.2 Timing information

As the computation of the remaining WCET at each observation point at run-time would generate prohibitive overhead to the critical task, we pre-compute at design-time partial WCETs by processing the ICFGs. This timing information is used at run-time to reduce the computation overhead. Compared to the theoretical results of [17], the computation of the timing information is extended and adapted to take into account the cost of the instrumentation of the critical tasks and the fact that more than one core runs critical tasks. Our WCET analysis is based on computing the remaining WCET from one observation point x until the end of a critical task τ_C, $\mathsf{RWCET_y}(x)$, where $y \in \{\mathsf{iso},\mathsf{max}\}$.

Isolated execution.

When a critical task has requested for isolated execution, p critical tasks are executed on the platform. The run-time control mechanism of the requester is not executed, as it has already requested isolated execution. Hence, the F_D is not called, whereas the F_E is called at the end of the critical task (Def. 3). For computing $\mathsf{RWCET_{iso}}(x)$, we only consider the feasible paths for the run-time control, that is when $C_{RT} = \mathit{false}$ and $\overline{C_{RT}} = \mathit{true}$. Using the remaining WCET analysis of an observation point x, we can compute remaining WCETs between an observation point x and its head point $\mathit{head}(x)$ for each critical task.

DEFINITION 5. *$d_{\mathrm{head}(x)-x}$ is the maximum time from* $\mathrm{head}(x)$ *to* x.

$$d_{\mathrm{head}(x)-x} = \mathit{RWCET_{iso}}(\mathrm{head}(x)) - \mathit{RWCET_{iso}}(x)$$

DEFINITION 6. *$w_{\mathrm{head}(x)}$ is the time between any two consecutive iterations j and $j+1$ of the* $\mathrm{head}(x)$*, when* $\mathrm{head}(x)$ *is the condition of a loop.*

$$w_c = \mathit{RWCET_{iso}}(c, j) - \mathit{RWCET_{iso}}(c, j+1), \forall j \leq n$$

To support the run-time $\mathsf{RWCET_{iso}}(x)$ computation, we store in memory $d_{head(x)-x}$ and the w_x for each point x, as depicted in Table 1.

Maximum load.

To guarantee that the critical tasks deadlines are always met, we must ensure that for each critical task, enough time is available to decide the suspension of the low criticality tasks at the next observation point. Hence, we apply our remaining WCET analysis to compute the $\mathsf{W_{max}}$ between any two consecutive observation points x, x' in each critical task in the maximum load scenario. Hence, the F_D is called at each observation point, whereas the F_E is not called (Def. 3).

$$\mathsf{W_{max}} = max_{x,x'}(\mathsf{RWCET_{max}}(x) - \mathsf{RWCET_{max}}(x'))$$

3. RUN-TIME

3.1 Global overview

At run-time, each critical task monitors its own execution and decides based on its local timing computation whether it requires the suspension of the low criticality tasks. In this case, it sends a request to the master. The local decisions per critical task are centralized and processed by the master which sends suspend or restart events to the low criticality tasks based on the received requests from the critical tasks.

When a critical task starts a new instance, it always assumes that the system is in maximum load, independently from the actual system status. This scheme is required for two reasons: firstly, when the critical tasks start their execution, they are not aware of the system status, which depends on the active requests made by the other critical tasks. This assumption guarantees that the overhead of our control mechanism is bounded, since a given critical task will generate at most two requests per instance. In a given time interval, the maximum number of requests depends on the periods and offsets of all critical tasks. Secondly, the remaining WCET of a critical task is computed when the run-time control is enabled, i.e. in the maximum load. Hence, when a critical task instance starts while the low critical tasks have been suspended, it still needs to execute its run-time monitoring control: in case the low criticality tasks are restarted during its execution, it must be able to compute its safety condition to guarantee a timely execution.

The master is responsible for serving requests and controls the suspension and the restart of the low criticality tasks. As our control mechanism is based on requests between 1) the cores that run the control mechanism through critical task instrumentation and the master, and 2) the master and the cores that run low criticality tasks, these requests are implemented by using the interrupts of the system and by developing the corresponding interrupt handling routines. The implementation of the the interrupts highly depends on the target platform. In this section we algorithmically describe the run-time control, whereas Section 4.1 presents the implementation of interrupts on our final platform.

3.2 Critical tasks

3.2.1 Monitoring ongoing execution

The monitoring of the ongoing execution highly depends on the target platform. During the implementation on the final system a set of low level functions are developed that

access the timing control registers of the target platform which provide access to the clock of the core, as described in detail in Section 4.1.1.

3.2.2 Computation of $\mathsf{RWCET_{iso}}(x)$

The algorithm for the computation in adynamic way of the $\mathsf{RWCET_{iso}}(x)$ of a critical task has been presented and its correctness has been proved in [17]. The code is depicted in Alg. 1. Briefly, when the ICFG is traversed in a forward direction, the remaining WCET is given by the remaining WCET of the head point c minus the time from the head point to the observation point x, $d[x] = d_{c-x}$. When the ICFG is traversed in a backward direction, the remaining WCET of this level is reduced by $w[x] = w_c$. An example of computing the $\mathsf{RWCET_{iso}}(x)$ for Fig. 4(b) is given in Table 2.

ALGORITHM 1: Computation of $\mathsf{RWCET_{iso}}(x)$

Pre-computed data: level, w, d, type
Input: x
Data: $o\_level = 0$, $ll = level[x]$, last_point[0]=*start*, $R[0]=WCET_{iso}$, $offset = 0$
Output: $RWCET_{iso}(x) = R[ll]$
if (type[x] ==*F_EXIT or F_ENEX*) **then** /* *condition 1* */
  o_level-=1
  $offset$ -= $level[x]$
$ll = offset + level[x]$
if $o\_level < ll$ **then** /* *condition 2* */
  $R[ll] = R[ll-1] - d[x]$
else
  if (*last_point[ll]* == x) **then** /* *condition 3* */
    $R[ll] = R[ll] - w[x]$
  else
    $R[ll] = R[ll-1] - d[x]$
last_point[ll]= x
o_level=ll
if (level[x] ==*F_ENTRY or F_ENEX*) **then** /* *condition 4* */
  $offset$ += $level[x]$

3.2.3 Safety condition

Per core that runs a critical task, at each observation point the control checks whether the low criticality tasks should be suspended through the safety condition given by Eq. 1. That is, for all tasks τ_{C_i}, at point x is:

$$\mathsf{RWCET_{iso}}(\tau_{C_i}, x) + \mathsf{W_{max}} + t_{\mathrm{SW}} \leq D_{\mathrm{C_i}} - \mathsf{ET}(\tau_{C_i}, x) \quad (2)$$

The suspension of the low criticality tasks occurs by the master when the safety condition of a critical task is violated. The overhead due to suspending and restarting the tasks, t_{SW}, includes the time to send the request to the master and the time to suspend the tasks. Hence, the theorem 1 proved in [17] also holds for the proposed distributed approach with several critical tasks.

Table 2: $RWCET_{iso}$ for the ICFG of Fig. 4(b).

Obs. Point	*condition* 1	2	3	4	Offset	$RWCET_{iso}$(x)	Last point[ll]	Obs. level
start	x	x	x	x	0	$R[0] = RWCET_{iso}$	LP[0]=0	0
$n_{0,a}$	0	1	x	0	0	$R[1] = R[0] - 0$	LP[1]=$n_{0,a}$	1
$f_{0,1}$	0	0	0	1	0	$R[1] = R[0] - d_{start-\mathrm{f_{0,1}}}$	LP[1]=$f_{0,1}$	1
$n_{1,a}$	0	1	x	0	1	$R[2] = R[1] - d_{\mathrm{f_{0,1}-n_{1,a}}}$	LP[2]=$n_{1,a}$	2
c	0	0	0	0	1	$R[2] = R[1] - d_{\mathrm{f_{0,1}-c}}$	LP[2]=c	2
$n_{1,b}$	0	1	x	0	1	$R[3] = R[2] - d_{\mathrm{c-n_{1,b}}}$	LP[3]=$n_{1,b}$	3
c	0	0	1	0	1	$R[2] = R[2] - w_{\mathrm{c}}$	LP[2]=c	2
$n_{1,b}$	0	1	x	0	1	$R[3] = R[2] - d_{\mathrm{c-n_{1,b}}}$	LP[3]=$n_{1,b}$	3
c	0	0	1	0	1	$R[2] = R[2] - w_{\mathrm{c}}$	LP[2]=c	2
$n_{0,b}$	1	0	0	0	0	$R[1] = R[0] - d_{\mathrm{start-n_{0,b}}}$	LP[1]=$n_{0,b}$	1

ALGORITHM 2: IHRs of the master

IHR_S{
num_requests=num_requests+1;
if *(num_requests==1)* **then** send suspend to low criticality tasks
}
IHR_E {
num_requests=num_requests-1;
if *(num_requests==0)* **then** send restart to low criticality tasks
}

THEOREM 1. *If $\forall i, \mathsf{WCET_{iso}}(\tau_{C_i}) \leq D_{C_i}$, then for any execution with the proposed run-time control, all τ_{C_i} respect their deadline.*

3.2.4 Low criticality tasks suspension

The node N_S of F_D sends a request to the master to suspend the low criticality tasks and stops its own control mechanism by setting the C_{RT} = *false*. Then, the master suspends the tasks by sending a set of interrupts to the corresponding cores. The implementation of the request is described in detail in Section 4.1.2.

3.2.5 End detection

The function N_E consists of sending a request to the master which notifies that the execution of the requester for isolated execution has finished. Then, the master decides over the restarting or not of the low criticality tasks. Similar to the low criticality task suspension, the implementation of the request depends on the target platform.

3.3 Master

The master accepts a set of requests from the N_S and the N_E of a critical task and sends a set of interrupts to the cores that run the low criticality tasks. We developed the corresponding Interrupt Handling Routines (IHR) for the master and for the cores that run the low criticality tasks, which serve these requests and the interrupts.

The master consists of two interrupt handling routines to serve the requests from the critical tasks, as shown in Alg. 2:

1. IHR_S is the interrupt handling routine called when the master receives a request from a critical task for the low criticality tasks' suspension. The master suspends the low criticality tasks when at least the safety condition of one critical core does not hold. Then:
   - it increases the number of active requests,
   - if it is the first received request, it sends the first interrupt to the cores that run low criticality tasks to suspend them.
2. IHR_E is the interrupt handling routine called when the master receives a request that notifies about the termination of the requester for isolated execution. Then:
   - it decreases the number of active requests,
   - if it is the last requester for isolated execution, it sends the second interrupt to the cores that run low criticality tasks to restart them.

Each core that runs low criticality tasks has an interrupt handling routine IHR_{LC} to serve the master interrupt:

- if it is the first interrupt received, an active polling mechanism takes place inside the interrupt handling routine,
- when the core receives a second interrupt, the polling is terminated and the IHR_{LC} finishes. Then, the low criticality task continues its execution.

4. EVALUATION RESULTS

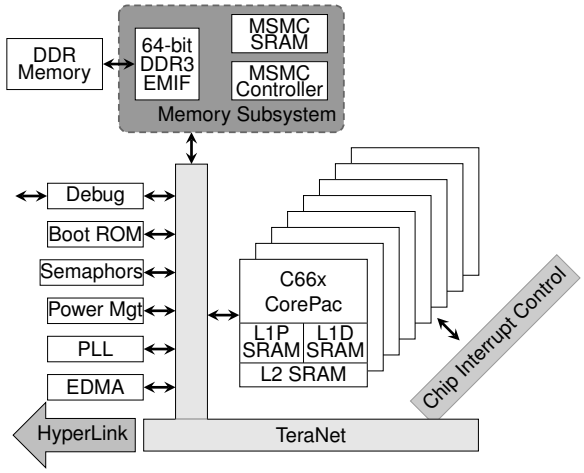


Figure 5: Overview of TMS platform.

We have implemented the proposed methodology in a real multi-core system, i.e. the TMS, to evaluate the behavior of our methodology. The implementation of two main mechanisms highly depends on the target platform: a) the time monitoring of the ongoing execution and b) the suspension/resuming of the low criticality tasks. These two mechanisms are described in Section 4.1. Section 4.2 presents the results under several experimental setups.

4.1 System Implementation

We are targeting a real multi-core COTS platform, i.e. the TMS320C6678 chip of Texas Instrument [16]. The platform is composed of 8 TMS320C66x DSP processors clocked at 1 GHz, which can issue up to 8 instructions in one clock cycle. Each core contains 32 KB level 1 program memory (L1P), 32 KB data memory (L1D), and 512 KB level 2 memory (L2) that may contain instructions and data, which can be configured as cache, SRAM or a hybrid. The level 3 memory (L3) of 4 MB on-chip SRAM and the external DDR3 of 512 MB memory is shared among the cores. The cores and the hardware modules are connected via the TeraNet on-chip network. The overview of the system is depicted in Fig. 5. We appropriately configure the TMS and implement the low-level functions to support the timing monitoring and the suspension/restart of the low criticality tasks. The overhead of our controller is: 70 cycles for the time monitoring, 200 cycles for sending an event and 501 cycles for computing the $\mathsf{RWCET}_{\mathsf{iso}}(x)$ and the safety condition.

4.1.1 Time monitoring

We reuse the bare-metal library of [13] that provides a set of timing functions to read the current clock by accessing the control registers TCSL and TCSH of the local core clock, which runs at the core's frequency. As the system should start the execution when all tasks have been loaded to the cores, this library also provides a synchronization scheme to ensure that cores start at the same time. When an observation point is reached, the run-time control uses the functions to read the real execution time of the system.

4.1.2 Suspension/Restart of low criticality tasks

The suspension and the resume of the low criticality tasks is implemented using the event and interrupt mechanisms of the TMS. The bare-metal library is extended with a set of functions that (1) configure the events and the interrupts of the TMS, (2) allow the use of the events by providing software setting, clearing and monitoring mechanisms for the events, (3) suspend or resume the low criticality tasks.

The TMS provides chip level events and core level events. A chip level event is created by the source core by setting the corresponding bit of the event to its Event Flag Register. Then, the Chip-level Interrupt Controller (CIC) is configured to generate host interrupts that act as core event inputs to the DSP interrupt controller of the destination core. The DSP interrupt controller allows up to 124 core events to be routed to the DSP interrupt/exception inputs. Each DSP can receive 12 maskable/configurable interrupts, 1 maskable exception, and 1 unmaskable interrupt/exception. When the DSP interrupt controller receives the core event, sets the corresponding bit to the Interrupt Flag Register of the destination core, an interrupt is generated to the core, the execution of the task is suspended and the Interrupt Handling Routine (IHR) is executed to deal with the interrupt.

In our implementation we have placed the master in the core 2, as depicted in Fig. 6, where core 0 and core 1 (gray cores) run critical tasks and the remaining cores run low criticality tasks (white cores).

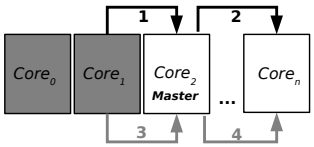


Figure 6: Suspend/restart of low criticality tasks

We have configured the TMS to map different chip level events for the synchronization between cores. Each chip level event is sent from a core to a host interrupt and then to a core level event for the corresponding DSP. The core that runs the master is configured to send a chip level event to each core that runs low criticality tasks to suspended their execution. Each core that runs a critical task sends a chip level event to the core that runs the master for the request to isolated execution and to notify that it has finished its execution. Then, we have implemented a core event configuration per DSP which maps each core event to one configurable interrupt. The master requests are handled by IHR_S and IHR_E and the cores with low criticality tasks by IHR_{LC}.

In our example in Fig. 6, in case the safety condition of one of the cores does not hold (e.g. core 1), the low criticality tasks must be suspended. Core 1 sends an interrupt (black arrow 1) to core 2 through node N_S to notify that suspension should immediately take place, it turns off its control mechanism and continues the execution of the critical task. When this interrupt is received at core 2, the execution of the low criticality task on core 2 is stopped and the IHR_S is executed. The IHR_S clears the received interrupt to be able to get the next one and increases the number of interrupts received. If it is the first interrupt received, it sends the interrupts to the remaining cores that run low criticality tasks (black arrow 2) by setting the corresponding bits to Event Flag Register of core 2. Then, it enters an active waiting mode until a new interrupt is received. The core level events are routed to the host interrupt controllers and

finally to the DSP interrupt controllers through the CIC. The corresponding bit to each Interrupt Flag Register is set, the execution of the low criticality task is stopped and the IHR_{LC} is executed to deal with the interrupt. It clears the interrupt to be able to get the next interrupt from the interrupt controller and puts itself in an active waiting mode until a new interrupt is received. In our example of Fig. 6, when the requester for isolated execution finishes (core 1), it sends an interrupt to notify core 2 (e.g. gray arrow 3). When the master in core 2 receives the interrupt, it reduces the number of active requests for isolated execution. If no other critical task has requested isolated execution, it sends the interrupts (e.g. gray arrow 4) to the cores that run low criticality tasks to notify that they can continue their execution and exits the active waiting mode. When the low criticality cores receive such an event, their IHR_{LC} exits the waiting mode and returns to the execution.

4.2 Experiments

In our experiments we have considered two loop and data dominated critical tasks with similar execution times and deadlines, i.e gemm from Polybench suite [25] which consists of three nested loops with main memory accesses and a set of arithmetic operations. The tasks are sharing the same main memory parts and run on core 0 and core 1. The remaining cores run a set of low criticality loop and data dominated tasks which are accessing different main memory parts. We explore the behavior of our methodology by tuning:

1. the application size of the critical task, which is given by the loop bounds and affects the arrays' size, the number of memory accesses and the execution time,
2. the number of cores that run low criticality tasks, i.e. from 1 core up to 6 cores,
3. the deadline of the critical tasks D_C, i.e. from tight deadlines close to the $\mathsf{WCET_{iso}}$ up to more relaxed deadlines,
4. the granularity of the software run-time control mechanism, i.e. the position of the observation points:
   - *coarse-grained*: At the head points of level 1 (HP1),
   - *fine-grained*: At the head points of levels 1 and 2 (HP2),
   - *very fine-grained*: At the head points of all levels (HP3).

4.2.1 Design-time analysis

Although we have configured our system to be supported by static WCET analysis, no static WCET analysis tool, such as AIT or OTAWA [28], is available for the TMS320C6678 platform. As it is not the scope of this paper to extend the supported platforms for static analysis tools, we use a measure-based approach by using the local timer of the cores that run the critical tasks to compute the timing information of the critical tasks, i.e. the $d(x)$, the $w(x)$ and the $\mathsf{W_{max}}$ for each critical task. The $d(x)$ and the $w(x)$ are measured when the critical tasks are the only tasks executed on the system and for each observation point of a critical task. To reduce the overhead introduced by our run-time measurements, we perform the timing measurements by considering one point per execution. For the computation of the $d(x)$, we use the maximum time observed between the head point and the point x. For the computation of the $w(x)$, we use the maximum time observed between any two consecutive iterations of the loop. The $\mathsf{W_{max}}$ is measured when the critical tasks run in maximum load for each different number of low criticality tasks and for each position of observation points. The computation of the $\mathsf{W_{max}}$ is performed by reading the local clock, subtracting any two subsequent observed times and using the maximum time difference.

The maximum time overhead of our run-time control for reading the local timer is 70ns, for the $\mathsf{RWCET_{iso}}(x)$ computation and the safety condition is 501ns and for the request and suspension/resume of the low criticality tasks is 200ns.

4.2.2 Behavior of high criticality tasks

Fig. 7 compares the $\mathsf{WCET_{max}}$ in maximum load with 6 low criticality tasks (6LC) in comparison with the $\mathsf{WCET_{iso}}$ depending on the application size given by the loop bounds. We observe that the difference between the $\mathsf{WCET_{iso}}$ and $\mathsf{WCET_{max}}$ increases when the number of congestions to the shared resources increases.

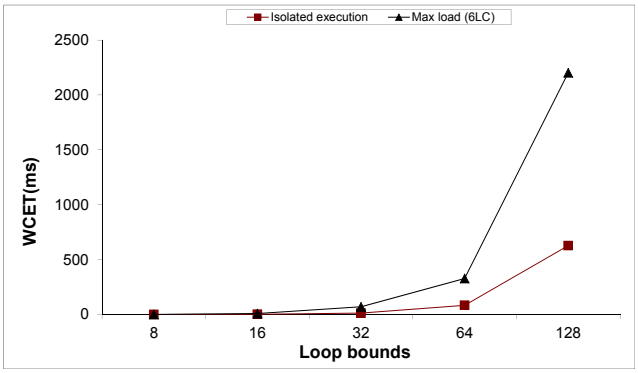


Figure 7: WCETs for several loop bounds

Fig. 8 depicts the $\mathsf{WCET_{max}}$ for several numbers of low criticality tasks (2 in which case $\mathsf{WCET_{max}} = \mathsf{WCET_{iso}}$ up to 8 parallel tasks) for an application size equal to 32. We observe that the WCET highly depends on the number of tasks that run in parallel. The WCET difference between isolated execution and maximum load with 6 low criticality tasks has a factor of 6.28.

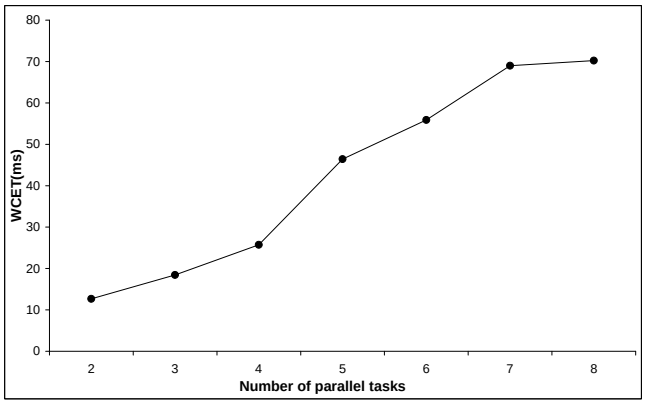


Figure 8: WCET for several parallel tasks

Fig. 9 illustrates the execution time of the critical tasks τ_{C1} and τ_{C2} for several deadlines and granularities for an application size equal to 32. We observe that the suspension of the low criticality tasks occurs:

- For the HP1 configuration: 1) relatively quickly, when the deadlines are tight (e.g. $<$ 14.00ms). The observation points are far away one from the other, the w_1 is quite

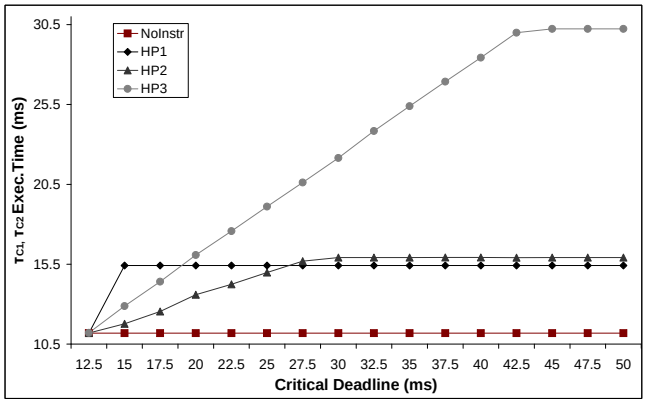


Figure 9: Exec.time of τ_{C1} & τ_{C2} for several deadlines

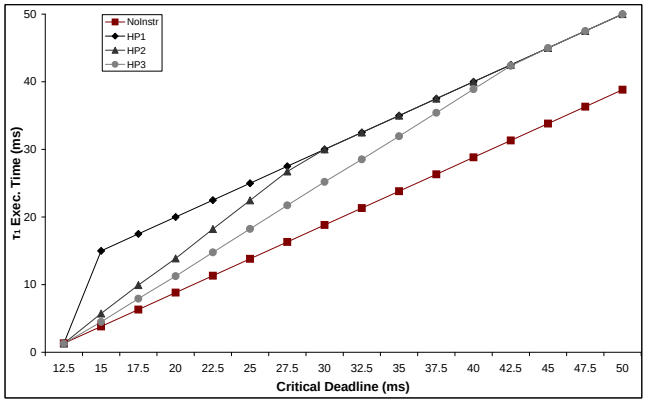


Figure 10: Execution time of τ_1 for several deadlines

large and thus the safety condition is early violated, and 2) never, when the deadline is relaxed. For instance, for deadlines > 14.00ms, the execution time of τ_{C1} and τ_{C2} is stable, as it has been completely executed maximum load scenario.

- For the HP2 and HP3 configuration: 1) quickly but later that HP1, when the deadlines are tight, as the lower granularities allow the exploration of the task suspension in smaller steps, 2) never, when the deadline is quite large. For these smaller granularities, the deadline in which the system is always executed in maximum load has a larger value that HP1 configuration, i.e. 30.00ms for HP2 and 45.00ms for HP3, and 3) in between otherwise.

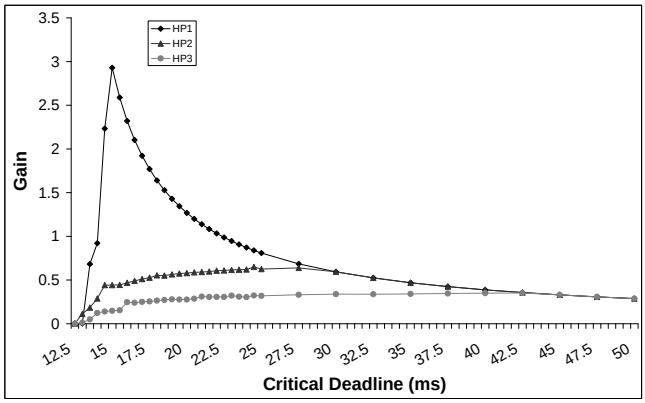
4.2.3 Gain on resources utilization

We explore the gains of our method by tuning (1) the deadline of the critical task and the (2) granularity of the observation points of our software controller.

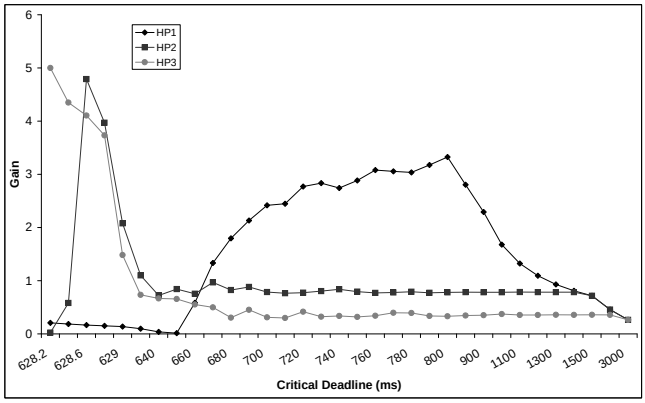
Fig. 10 illustrates the execution time of the low criticality task τ_1 for several deadlines and granularities for an application size equal to 32. We observe that the more time the critical task spends in the maximum load, the longer are the execution times of the low criticality tasks. To explore the gain of our methodology, we consider the following notion:

DEFINITION 7 (RELATIVE GAIN). *Let t denote the execution time of the low criticality task with our methodology and t_{iso} the execution time in isolation mode. The* relative gain *obtained by the methodology is:*

$$(t - t_{iso})/t_{iso}$$



(a)



(b)

Figure 11: Gain for several deadlines with application size a) 32 and b) 128

Fig. 11(a) presents the behavior of the relative gain of our approach with the HP1, HP2 and HP3 configuration with respect to the different deadlines for an application size equal to 32. From the experiments, we observe that our approach achieves significant relative gains:

- For the more relaxed deadlines (still below the $\mathsf{RWCET}_{\mathsf{max}}$ i.e. 18.50ms), e.g. in 15.5ms, we have the highest gain of 2.9885 or 298.85 % for HP1. The HP1 configuration provides the largest gain. It calls the run-time controller in larger steps, and the slack time, i.e. the time between the critical task's end and the deadline, is quite short.
- For larger deadlines, we still have a gain which is decreasing when the deadline is further relaxed.
- For tight deadlines, e.g. <13.00ms we achieve a relative gain of 11.27% for the fine grained and a relative gain of 4.93% for very fine grained granularities.

By increasing the application size to 128, we observe that the gains of the lower granularities are increased. Fig. 11(b) presents the behavior of the relative gain of our approach with the HP1, HP2 and HP3 configuration with respect to the different deadlines. We observe that:

- For tight deadlines (e.g. <660ms), the HP1 configuration has a small gain from 2.5%-23.89%, as the w_1 is significantly increased and the critical task requests for isolated

execution at the first observation point. However, the smaller granularities achieve significant gains because the observation points are placed closer. The HP2 configuration achieves a relative gain of 479% in 628.6ms, whereas for even tighter deadlines (e.g. 628.2ms), the HP3 configuration achieves a gain of 556%.

- For more relaxed deadlines (e.g. >660ms and <2000ms), the gain of the HP1 configuration is increased up to 317%, as the time slack is increased and no need exists to explore the suspension of the low criticality tasks in smaller steps. The gains of the HP2 and HP3 remain stable and equal to 78.5% and 37.5%, respectively, as the gain due to the time slack increase balances the loss of the controller overhead.
- For larger deadlines (e.g. >2000ms), we still have a gain which is decreasing while the deadline is further relaxed.

5. RELATED WORK

This section briefly presents the different approaches on the mixed-critical systems, whereas a detailed survey on the mixed-criticality research up to now is available in [10].

5.1 Task scheduling

The majority of mixed-criticality scheduling work has been mainly addressed for uni-processor platforms (e.g. [27, 5, 4, 8]), which is not directly applicable in multicore platforms. In the latter, shared resources exist and time compositionality cannot be ensured [11], as the WCET analysis cannot be applied independently per task.

In multicore platforms, several approaches exist that assume that the task set is schedulable at least at the high criticality level. For instance, in [7], both hard real-time and soft real-time tasks are scheduled using an Earliest Deadline First for Hard real-time, Soft real-time and Best effort tasks (EDF-HSB) approach with the assumption that the hard real-time tasks are statically schedulable. When time slack occurs at run-time, it is reallocated to non hard real-time tasks. Another example is the two level mixed-criticality scheduling for multicore platforms proposed in [1] and extended in [21], where the tasks are scheduled based on the WCET of their criticality level and the time slack is reallocated to lower criticality levels. The tasks of different criticality levels are scheduled with different appropriate scheduling approaches. The tasks with the lowest criticality level are allowed to be executed when no higher criticality task is running, i.e. in the critical tasks are executed in isolation. In addition, several mixed-critical scheduling policies have been implemented in the LITMUS^{RT} framework [15].

Less pessimistic approaches, such as [19, 3, 24], use several WCETs per task during task scheduling. Initially, all tasks are assigned their low criticality WCET, which is a less pessimistic bound on WCET given by designers. This WCET derives from a set of test cases [9] and is the maximum execution time observed during execution of the system. The proposed algorithms in [19, 3] describe a generalization of the preemptive uniprocessor algorithm EDF with Virtual Deadlines (EDF-VD) to multiprocessor platforms. At run-time, they observe if the tasks have signaled termination at their low criticality WCET. The check for the switching to higher criticality WCET occurs once in a pre-defined position based on the low criticality WCET. If no signal termination exists by that time, the criticality level of the tasks is increased and the tasks with lower criticality levels are dropped. This occurs because a scenario of higher criticality is now considered and the completion of jobs of lower criticality becomes irrelevant for the new scenario [2]. Further extensions of similar methods are presented in [14] which avoid the abandoning of the low criticality tasks during high criticality mode and return to the low criticality mode after the high criticality mode has been terminated. In [24] a Mixed-criticality Scheduling on Multiprocessor (MSM) algorithm is proposed which uses a global fixed priority based approach. When the switching of criticality level occurs, the low criticality tasks are dropped. The approach presented in [6] considers mixed-critical systems and time-triggered paradigm where WCET estimates are allowed to overrun. A run-time monitor is in charge of detecting these overruns and switching on a schedule selected from a set of pre-computed schedules.

Our approach is orthogonal to the aforementioned approaches: we consider two different types of WCETs, which are not based on their reliability like existing methods but on how the critical task is executed on the platform, i.e. in maximum load or in isolated execution. Our WCET estimations in both scenarios are safe. Our method is applicable in cases where the system is considered as unschedulable, e.g. when the values of low and high criticality WCET are estimated above the deadline of the critical task. We also perform a switch which suspends the low criticality tasks in risk of a local overrun of the critical task. However, the time instants where the switching can occur and the guarantee of meeting the deadline of the critical task are formally defined and proved. Then, the switching is decided at run-time by exploring the ongoing execution of the critical task. A brief description of the general idea is detailed in [18] and the formal description and the proof in [17]. In this paper, we extended our approach by increasing the number of critical tasks which have now different deadlines, periods and offsets and the number of cores where the critical tasks run, by refining the observation points to insert instrumentation and by implementing our controller in a real multicore platform.

5.2 Run-time control implementation

Several approaches propose resources reallocation based on information derived from monitoring their utilization, e.g. the memory accesses. For instance, in [23] *interference-sensitive* WCETs are computed based on a preliminary analysis of the resource usage of tasks. The shared resources are off-line partitioned among tasks. A run-time monitoring device observes the resource usage of each task and suspends the task that overtakes the allocated capacity. In [22] the approach is extended by allowing safe dynamic changes in the resource partitioning, when resources are underutilized. In [30] an approach has been developed to reserve memory accesses for critical tasks. A run-time controller has been implemented which regulates the accesses to the shared memory and ensures temporal isolation among tasks. An off-line profiling technique has been proposed in [20] which finds the most frequently accessed memory pages in a task. Then, this information is used to modify the variables' position in the shared caches in order to reduce the interferences. Another hardware approach is described in [12] where the monitoring is only performed when enough slack time exists which guarantees that the monitoring does not impact the meeting of the real-time constraints of the tasks. If the slack is insufficient, a dropping operation is executed to minimize the monitoring overhead.

In contrast, our approach is not based on monitoring the

accesses to the shared resources, but on monitoring the ongoing execution time of the critical tasks. In case the time to reach an observation point is too high, this implicitly means that the low criticality tasks have generated many contentions to the shared resources. If tolerating more interferences could cause a dangerous slow down of the critical task, the switching to isolated execution is mandatory.

6. CONCLUSION & FUTURE WORK

In this work, we present a methodology which improves the resources utilization by increasing the task parallelism, while guaranteeing the real-time response of the critical tasks. These tasks are described by a set of ICFGs and the structure and time analysis is applied to compute the required data for the run-time part. At run-time, a low-overhead controller per critical task computes the remaining WCET and decides the switching to isolated execution, which is executed by the master entity. We have implemented our approach on a multi-core COTS and we observed gains up to 556% for our case study.

As future directions, we will apply our approach to other types of critical tasks to obtain a global view of the gains of our method. The next major challenge to solve is the development of a methodology to decide the position of the observation points over the ICFGs. We would like to explore the effect of the different scheduling techniques on our approach. We would like also to explore the case where in isolated execution we allow some low criticality tasks to run on other cores even if some active requests exist. In addition, we consider the extension to several criticality levels by adapting the computation of the partial RWCETs and the definition of the isolated execution by exploring several cases regarding the parallel execution of tasks during isolated execution. We believe that the combination of our approach with time and partitioning methods will further increase the task parallelization during isolated execution.

7. REFERENCES

[1] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. Multicore operating-system support for mixed criticality. In *WMC*, April 2009.

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. L., A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *Trans. Computers*, 61(8):1140–1152, 2012.

[3] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, pages 1–36, 2013.

[4] S. Baruah, L. Haohan, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, pages 13–22, USA, 2010. IEEE.

[5] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, USA, 2008. IEEE.

[6] S. K. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS*, pages 3–12, 2011.

[7] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *ECRTS*, pages 61–70. IEEE, 2007.

[8] A. Burns and B. Baruah. Towards a more practical model for mixed criticality systems. In *RTSS*, 2013.

[9] A. Burns and S. Baruah. Timing faults and mixed criticality systems. In C. Jones and J. Lloyd, editors, *Dependable and Historic Computing*, volume 6875 of *LNCS*, pages 147–166. Springer Berlin Heidelberg, 2011.

[10] A. Burns and R. Davis. Mixed criticality systems - a review. Technical report, Department of Computer Science, University of York, York, UK, 2014.

[11] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Ingénieurs de l'Automobile*, 807:36–42, September 2010.

[12] T. C. Daniel Lo, Mohamed Ismail and G. E. Suh. Slack-aware opportunistic monitoring for real-time systems. In *RTAS*, USA, 2014. IEEE.

[13] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *ERTS*, 2014.

[14] T. Fleming and A. Burns. Extending mixed criticality scheduling. In *RTSS*, 2013.

[15] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *RTAS*, pages 197–208, 2012.

[16] T. Instruments. TMS320c6678 Multicore fixed and floating-point digital signal processor. Technical Report SPRS691D, TI Incorporated, 2013.

[17] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, and M. Roy. Run-time control to increase task parallelism in mixed-critical systems. In *In 26th Euromicro Conference on Real-Time Systems (ECRTS'14)*, 2014.

[18] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, M. Roy, and F. Vargas. Monitoring on-line timing information to support mixed-critical workloads. In *WiP RTSS*, 2013.

[19] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *ECRTS*, pages 166–175, 2012.

[20] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS*, pages 45–54, 2013.

[21] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *CIT*, pages 1864–1871, 2010.

[22] J. Nowotsch and M. Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *RTNS*, pages 151–160, 2013.

[23] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. Technical Report 2013-10, University of Augsburg, Germany, 2013.

[24] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *ECRTS*, pages 309–320, USA, 2012. IEEE.

[25] L.-N. Pouchet et al. Polybenchmarks benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2013.

[26] SAE. Aerospace recommended practices 4754a - development of civil aircraft and systems, 2010. SAE.

[27] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, USA, 2007. IEEE.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM TECS*, 7(3), 2008.

[29] R. Wilhelm and J. Reineke. Embedded systems: Many cores - many problems. In *SIES'12*, pages 176–180, 2012.

[30] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, pages 55–64, 2013.